CHAPTER

1

Principles of SOA

The physician can bury his mistakes, but the architect can only advise his client to plant vines.

Frank Lloyd Wright (1953)

COPYRIGHTED MATERIAL

Computer systems are critical for modern, increasingly global businesses. These organizations continually strive for shorter time to market and to lower the cost of developing and maintaining computer applications to support their operations. However, according to regular reports from the Standish Group between the mid-1990s and the present day, around two thirds of large US projects fail, either through cancellation, overrunning their budgets massively, delivering a product that is never put into production or requiring major rework as soon as delivered. Outright project failures account for 15 % of all projects, a vast improvement over the 31 % failure rate reported in the first survey in 1994 but still a grim fact. On top of this, projects that were over time, over budget or lacking critical features and requirements totalled 51 % of all projects in the 2004 survey. It is not incredible to extrapolate these scandalous figures to other parts of the world. What is harder to believe is that our industry, and the people in it, can remain insouciant in the face of such a shameful situation. Clearly we should be doing something differently.

1.1 Why Projects Fail

The Standish conclusions are further illuminated by the data represented in Figure 1-1, which shows the fate of US defence projects (Connell and

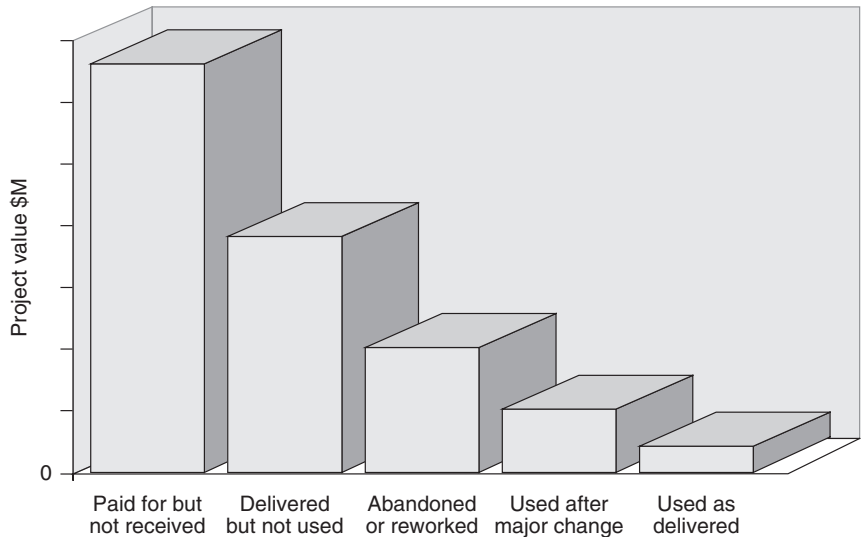

Figure 1-1 The outcome of US defence projects according to US government statistics.

Shafer, 1989). It must be remembered that these systems were mainly mainframe systems written in languages such as PL/1 and COBOL and it may be unfair to make a comparison with systems developed with modern tools. However, the point that something was wrong, even back then, cannot be avoided. Furthermore, the modern evidence seems to suggest that, sadly, not that much has changed.

The Standish surveys also looked into the reasons why people involved in the sample projects thought such projects fail. The reasons given included, *inter alia*:

- lack of user involvement;
- no clearly stated requirements;
- absence of project 'ownership';
- lack of clear vision and objectives.

Why should this be? If the cause really is lack of user involvement (leading to the other three) then we must ask why users are so reluctant. If the system is worth building (and paying for) then, surely, it must be worth spending some time to ensure it does what the user really wants. Is it because they have had bad experiences with IT in the past, perhaps?

Could it be that previous projects involved copious amounts of time spent with that clever C++ programmer (you know, the one with the Ph.D. in Arrogance) poring over huge diagrams that obviously made some sort of sense to him? Could it be that, by the time the system was delivered, the

business had moved on so that changes had to be made and these changes took forever and ramped up the cost 'olympically'. Of course I'm too busy!

There are several reasons why our customers are exasperated with us nice IT folk.

- The typical IT person is more concerned with specification than with modelling requirements. The project manager wants to rein in everything to inside the system boundary and the designers think that a sexy system architecture is cool.
- IT folk do not speak the same language as their users. We speak UML and you, Mr User, must learn it if you want us to be able to communicate successfully.
- The architecture of our systems is driven by fashionable technology and short term project goals. This means that the principles of software engineering best practice are usually ignored in the scramble to cut and test the code. This accounts for the discrediting of the various volleys of silver bullets over the past years: structured design, object-oriented methods, component based development, etc. SOA, in some ways is a repackaging of the same (good) ideas – as we will see in the next chapter.
- Furthermore, the level of abstraction at which we work tends to be far too low.

IT departments are often culturally and technically miles away from the concerns and thought processes of the customers they serve. The problem is, thus, far broader than the need for SOA or any other technological solution; the real problem we have to solve is how to align IT practice with business need and begin to speak a common language. If this can be achieved there is a chance the SOA will not suffer the ignominy of its illustrious predecessors.

1.2 Aligning IT with Business – Speaking a Common Language

To believe that adopting the latest technology fad, be it service oriented architecture, business process modelling or even a business rules management system will, on its own, solve this problem is nothing short of naïve. To align IT with business we must consider all these along with innovative approaches to requirements engineering and system modelling.

The problem of requirements engineering is a modelling problem. We must model the business: its processes, its goals, its systems, its people and structure and even its culture. But we also need to model potential

solutions: in this context, networks of loosely coupled services (applications) that make sense to the business and contribute to its goals. Ideally, the services (the technology, if you must) will map clearly onto the business needs and processes. This implies that we need a language rich enough to describe both the business and its systems and, more importantly, a language that can be understood by all stakeholders.

If one adopts the common misconception that understanding a client's requirements is the same as specifying a system that will meet those requirements, one can then blithely infer that use case analysis is the only requirements modelling technique needed. Jackson (1998) pours scorn on this idea, arguing that use cases are useful for specifying systems but that they cannot describe requirements fully. Use cases connect actors – which represent *users* adopting rôles – to systems. Requirements, on the other hand, may be those of people and organizations that never get anywhere near the system boundary.

In Figure 1-2 we see a depiction of part of Jackson's argument. A requirements document must be written in a language whose designations concern things in the world in which the system is embedded (including of course that system). Specifications need only describe the interfaces of the system and therefore depend on different designations. A specification *S* describes the interface of phenomena shared between the world and the system; use cases may be used to express these. A requirements model *R* is a description over these and other phenomena in the world. *R* depends on both the specification *and* the world. Jackson also states that 'the customer is usually interested in effects that are felt some distance from the machine'.

Ignoring non-user interactions can lead to us missing important reengineering opportunities. I once worked on a rule-based order processing and auto-pricing system, whose aim was to take orders from customers electronically and price them automatically using various, often complex, pricing engines. The problem was that some orders were too complex or too large to admit of automatic handling. These had to be looked at by a salesman who would of course have an interface with the 'system'. So

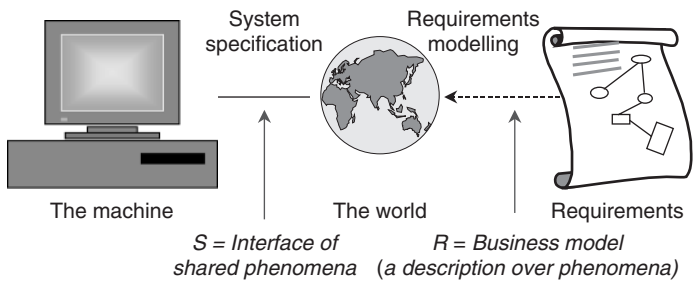

Figure 1-2 Specifications are not requirements models.

far, so good: a rule engine would screen 'illegal' or 'handle manually' orders. The salesman would then apply his various spreadsheets and other routines to such orders. But a further problem existed: some orders were so complicated as to be beyond the skills of the salesman, who did not have expertise in financial mathematics. For these orders, the salesman had to go across the office and talk to a specialist trader. He/she did have the requisite Ph.D. in Financial Engineering. We also modelled this conversation as the 'pseudo-use-case' shown as '`Help!`' in Figure 1-3 and, as a result, when our domain expert looked at the simulation we had built, he/she realized immediately that if we gave the trader a screen (for orders re-routed) we could radically improve the workflow, and thereby customer service. Even this relatively minor excursion away from the system boundary thus had a big cash import because the orders for complex products were precisely the most profitable orders – by a long way. In many, more complex cases, the importance of going beyond the boundary will be greater still. Jackson gives an example of patient monitoring wherein sensors are attached to a patient's vital signs, and alarms (i.e. variances from tolerance) are forwarded to a nurse's workstation. The problem is that, should an alarm be triggered, the nurse is not normally the actor who will save the patient's life. She must run down the corridor to fetch a doctor. Thus, the critical use case is not at the system boundary and would be ignored in the conventional approach. Put starkly, concentrate on the use cases at the system boundary and people may die.

My interpretation of Jackson's argument is that we need a specific technique for modelling business processes distinct from, but compatible with, use case models of specifications. The alternative is to fall back on a veritable 'Russian doll' of nested models described in terms of 'business

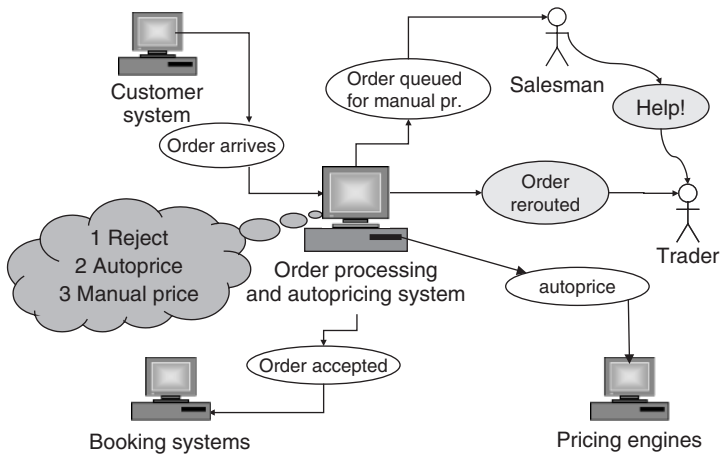

Figure 1-3 A process for order processing.

use cases' (Jacobson *et al*., 1995): an approach that is not only clumsy but fails to address the above arguments. Thus, we need to know the answers to the following two questions before we can proceed.

- What is a model?
- What is a business process?

We defer dealing with the second question until a later chapter. Later in this book we will explore various notations and approaches to business process modelling. Let us first look at the nature of models.

1.2.1 Models

Modelling is central to software engineering practice and especially within the context of service oriented architecture. A **model** is a representation of some thing or system of things with all or some of the following properties.

- It is always *different* from the thing or system being modelled (the *original*) in scale, implementation or behaviour.
- It has the shape or appearance of the original (an iconic model).
- It can be manipulated or exercised in such a way that its behaviour or properties can be used to predict the behaviour or properties of the original (a simulation model).
- There is always some correspondence between the model and the original.

Examples of models abound throughout daily life: mock-ups of aeroplanes in wind tunnels; architectural scale models; models of network traffic using compressed air in tubes or electrical circuits; scaled models of silting up in river estuaries; software models of gas combustion in car engines. Of course, *all* software is a model of something, just as all mathematical equations are (analytic) models.

Jackson (1995) relates models to descriptions by saying that modelling a domain involves making designations of the primitives of the domain and then using these to build a description of the properties, relationships and behaviour that are true of that domain. For example, if we take the domain of sending birthday cards to one's friends, we might make designations:

```
p is a friend;
d is a date (day and month);
B(p,d) says that p was born on d.
```

Then we can make descriptions like: for every p, there is exactly one B. Jackson suggests that modelling is all about ensuring that the descriptions

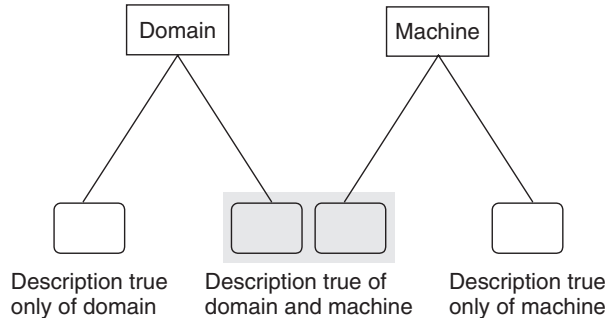

Figure 1-4 M is for model (after Jackson, 1995).

apply equally well to the model and to the original domain. In the context of computer models this might mean that the instances of a class or the records of a database are made to correspond uniquely to domain instances of our friends. Most usefully, Jackson presents this concept as the M configuration shown in Figure 1-4.

The Domain and the Machine are different; in the domain friends do not reside in disk sectors. There are many things in the domain that are not in our model, such as our friends' legs or pimples. There are also things that go on in computers that we are not concerned with in a model, such as load balancing. The model comprises the features shared between the domain and the machine.

This understanding of what a model is can be applied to the problem of service modelling. We must understand clearly that a service is both a model of the domain and a potentially implementable machine model. But we must begin with a model of the domain to understand and validate the requirements.

This understanding leads also to another common difficulty with computer systems: they can get out of synch with the world. The most topical example of this is probably identity theft; the computer model thinks you live somewhere else! It is therefore a key requirement of most systems that there should be a mechanism for re-synching the models should they become out of kilter.

Good models are abstract but 'real'; they ignore unnecessary detail but correspond exactly to the true state of the business. Good models are re-synchable. Good models are understandable to non-technical people.

Applying this understanding to modelling services, a service is derived from a model of the business domain and of the potential implementation domain. Defining the services precisely is the only way we can be sure that they meet current needs. Modelling unifies and clarifies those services and allows us to know what they are. Precise definition also provides a contract for the software developers.

Note that services may have relationships between them and often do: this service can only be accessed after that one; these services can be done in parallel; this service enables or disables that one; etc.

These relationships are about business rules. If we change the relationship, we will change the way we do business or re-engineer the business. Such an approach helps us to integrate different parts of the business and be more agile. It also supports greater reuse because we can know exactly what the service is. Additionally, it helps identify services required from other businesses that, hitherto, we might have been forced to create from scratch, thus duplicating them.

In later chapters we will return to the question of how to construct a sufficiently rich language for business modelling that is equally understandable to the user and to IT staff. We will focus on techniques for modelling requirements, system and business processes specifications on the way to delivering an SOA that can support them properly and allow for flexible and low-cost evolution. But first we must understand the principles and basic concepts of service oriented architecture.

1.3 What is Service Oriented Architecture?

Service oriented architecture (SOA) is an architectural concept in software design that emphasizes the use of combined loosely coupled services to support business requirements directly. In SOA, resources are made available to service consumers in the network as independent artefacts that are accessed in a standardized way. This adherence to standardization is definitional. SOA is precisely about raising the level of abstraction so that requirements and business processes can be discussed in a language understood by business people as well as IT folk.

The main idea behind SOA is the desire to build applications that support a business process by combining a number of smaller 'business services' into a complete business process or workflow. Each of these services is a stand-alone piece of software providing business functionality that is loosely coupled to the other services (other pieces of software) which make up the application. Examples of a business service could be checking details about a customer, validating a customer payment, sending an invoice to a customer, synchronizing or transferring data between systems, or converting a document from one format to another. Many of these services will be particular to a business; however, some will also be standard services that could either be purchased as software or will be readily available on the internet in the form of web services. New services can also be created from existing applications or by writing new ones using your preferred development framework.

Many definitions of SOA identify the use of web services (using SOAP and WSDL) in its implementation; however it is possible to implement SOA using any service-based technology. Though built on similar principles, SOA is *not* the same as web services. SOA is independent of any specific technologies.

A software architecture is a representation of a software system – how all the pieces fit together. It describes the most effective way to design the system within a set of constraints or a defined infrastructure. An architectural style is a family of architectures sharing common themes and a recognizable common vision, in the same way that we can recognize the shared vision of Gothic or Georgian architecture. Service oriented architecture is an architectural style whose goal is to achieve loose coupling among interacting software agents. A software agent is an application, a piece of software, a component, a program, etc; also known in older terminology as a module – a piece of software that does work. A **service** is a unit of work carried out by a **service provider** to achieve some desired result for the **service consumer**. Typically, a unit of work would be some sort of (business) transaction. The service consumer has a goal in mind, and the task of the service provider is to achieve that goal on behalf of the consumer or help the consumer to do so. Both provider and consumer are rôles played by (possibly software) agents.

Although a service is just an interface, it will thus be implemented by a software component, or collection of software components, that implements a reusable business function, such managing customers or managing customer accounts. Services are defined by their interfaces, which describe what the services can do. How the service works is hidden inside the component or components that provide the service.

Service-oriented architecture is an approach for designing and building applications by constructing them from loosely coupled services (existing services where possible) and discovering and writing new services as necessary.

The SOA approach encourages loose coupling between the services that make up an application; contrast this to traditional monolithic architectures, which are characterized by tight interdependence between the parts of an application.

Figure 1-5 illustrates how a user might interact with a product ordering or quotation service. The user asks a question and gets a straightforward and useful answer. To assemble this response the system actually relies on four lower level services, including one which could well be rule-based, since tax regulations vary quite often.

The most important properties of a service are as follows.

- Every service has a **contract**: a description of what the service will do for the user and what it requires of the environment and indeed the

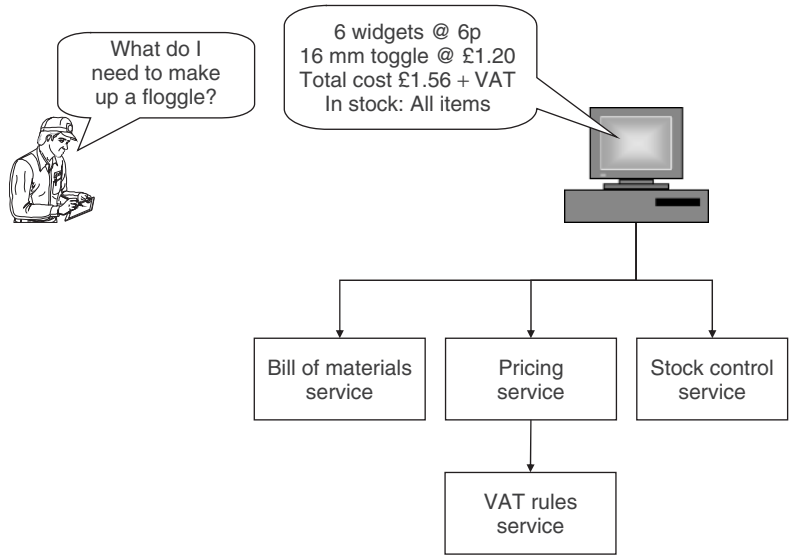
What do I need to make up a floggle?
6 widgets @ 6p
16 mm toggle @ £1.20
Total cost £1.56 + VAT
In stock: All items
Bill of materials service
Pricing service
Stock control service
VAT rules service

Figure 1-5 Composing loosely coupled services.

user. This is what enables the loose coupling between services and is the basis for 'discovering' [services over a distributed environment]. Clearly defined contracts are essential if we are to achieve composition and reuse of services.

- Services can be **discovered**. They are designed to have a description in a directory, so that they can be found and accessed via a discovery mechanism. This too helps make services more readily reusable. Discovery is a property that can be given to any piece of software – but it doesn't necessarily mean anyone will find that software actually useful.
- Services are **abstract** – the only part of a service that is visible to the outside world is the service description (the contract). This contributes greatly to service reusability or sharing.
- Services are **autonomous**: they have control over the logic they encapsulate; they decide how any arriving messages should be processed or forwarded. This enables service composition and makes reuse easier to achieve.
- Services can be **composed** – they can be combined with other services to satisfy a set of business requirements, to solve new problems.
- Services should be **loosely coupled** to other services. This enables composition and encourages autonomy.
- Services are **stateless with respect to complete transactions**. Services minimize the storage of information specific to an activity (a use of

the service). This too helps with composability. A stateless service is an ideal that should always be strived for, but we can weaken this requirement by passing the state either directly or indirectly with any message exchange.

- Services should be **reusable**. Systems are best divided into services with the conscious intention of promoting reuse.

One needs to focus on precisely these eight properties of a service when developing a new service or identifying and wrapping existing applications as services. Looking closely at these, we can see that reuse depends on all the other properties.

- Abstraction helps with packaging for reuse.
- An autonomous and stateless service is more likely to be reusable.
- Loose coupling means minimizing dependencies on other software and this encourages reuse.
- A service that is discovered can be used, and used again; it can be reused.

Also, adversely:

- Reuse takes place when a service is composed with other services.
- A service that is autonomous and stateless is easier to compose with other services.
- It is precisely loose coupling that allows composition.

Thus both reuse and composition result from the other six properties.

There is an even more important property that we have not listed here. As shown in Figure 1-6, a service must supply a service to the business. This means that the service does something that achieves a business goal. It will be about a sale, identifying a customer, reserving a seat on an aeroplane or some such. The service will not be about supporting functions in a user interface.

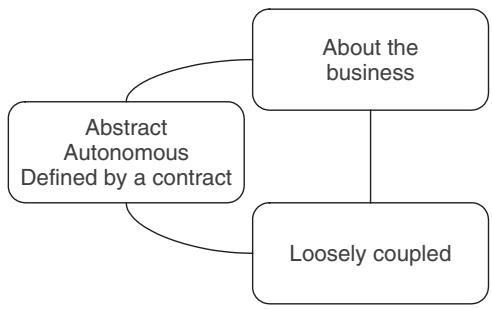

Figure 1-6 The key properties of a service.

The interface of a service, as presented to the user, displays a set of operations that make up that service. These operations are about supporting the user of the service. But who *is* the user of the service? The usual answer is an actor (using UML use case terminology). An actor is a user adopting a rôle or another system that interacts with the service. However, it is necessary to think more carefully about this. Services are about supporting the business, not supporting a computer user or another system. The service should support something or somebody *in the business*, in the real world: the **real user** of the service. The real user, in this sense may be someone who never comes anywhere near a computer or browser; someone who is far away from the system boundary.

The operations of a service should be abstract and at a high level of abstraction and be about the business. Some examples will illustrate this point.

When reserving a hotel room, say, we need to create a record with all the necessary details obtained from the real user, the person making the reservation; rather than

```
get guest details (name, address, contact tel)
get reservation details (room type, arrival date, days)
get reserver details(name, address, contact tel)
```

These are probably operations supporting a user interface or procedure calls in some programming language. Instead of this, a service invocation (message) should contain all the information that is needed for a coherent transaction to be possible. For example:

```
message<guest name, guest address, guest contact
tel, room type, arrival date, days, reserver name,
reserver address, reserver contact tel>
```

SOA emphasizes building systems for the user, not systems for the IT department: systems for the benefits of employees, customers, suppliers, partners; systems for the 'real' user!

In the past, clerks used 'terminals' to access 'batch systems'; now customers use the web . . . or do they?

I was once involved in the design of a web radio station. Our approach soon teased out the fact that the most important business objective behind the initiative was neither to do with entertainment, any Reithian concern with edification nor with selling advertising (they would then have only lost the same revenue from the airtime ads). No, the main purpose was to sell CDs. The target audience was teenagers, so the strategy included providing lots of contents concerning current pop stars and bands. Here is one of the scenarios we explored during one requirements workshop.

Denise Green is visiting her friend Tracey Black. They are eating chocolate and discussing music and decide to log on to the station to check out some

groovy sounds and get all the latest poop on their favourite boy bands. To their delight, BoyzInCustard have a new album out today. They *must* have it. So far so good. But there is a problem; neither girl is old enough to have a credit card. Never mind, good old Mum's got one. Mrs Black wouldn't go near a computer if her life depended on it. However, she is precisely the real user so far as the financial part of the radio station's business process is concerned.

Figure 1-7 provides another way to visualize this point.

Loose coupling is about having a simple interface with very low dependency – an interface that mandates collaboration with 50 other services in order for it to do its job and for them to do their jobs would be impossible to extract for reuse. To say that services can be loosely coupled implies that the service interface must be independent of the implementation. There can be unintended consequences of coupling (both good and bad ones). Ideally, services are 'stateless'; which means that, when a message arrives at a service interface, the service processes the message and returns an answer or result to the consumer; it remembers nothing about the message afterwards; it does not even need (apart from the reply address) to know anything about the consumer.

A service can receive a message, process it, return a reply and then forget all about it. At any rate, this is all we can see from the outside. Looking inside the service, however, we might see three types of communication ports or 'endpoints'.

- Ports for messages entering the service; the **entry** ports.
- A departure port for sending the reply message; the **exit** port.
- A **rejected message** port for invalid messages.

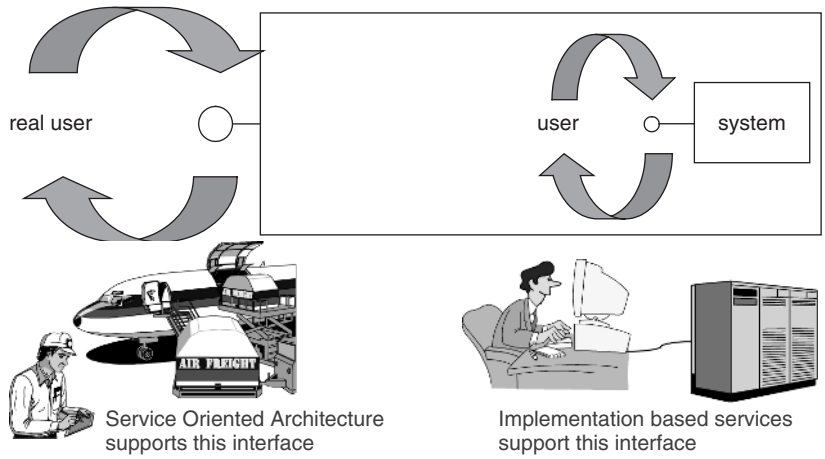

Figure 1-7 Services for customers not clerks.

There could be more ports such as those for queuing application faults or tracking messages, but the above three are the basic ones.

A service will read a message from the entry port, process it and write the reply to the exit port. If the message is invalid (it is not recognized by the service), it is written to the rejected message port. Developing a service in this style is reasonably straightforward. The service knows nothing about the outside environment, it is entirely self-contained and all it needs to do is process a message it recognizes, reject a message it doesn't, send any reply as necessary, and then forget about the message.

Note that this type of service will probably need a complex environment it can exist in. The trick is to make this complex environment easy to configure and use. As we shall see later, enterprise service bus products offer one such configurable environment.

In summary, the perfect service is a black box, loosely coupled to all users. Its overall action is input–process–output, with no memory of the input after producing the output.

It is vitally important that a service can be extended. Businesses change rapidly and so it is important that software changes to reflect this. This will involve changes to service providers, service consumers and the messages. If these changes cannot be made, then everyone is locked into the current version of a service, it will not be possible to extend it to reflect the new business opportunities. What changes can be made to a service without invalidating it for current users?

We can change the implementation providing the interface is still satisfied, we can change the interface by adding function of accepting more types of input and we can add fields to a message. Changes other than these are likely to invalidate or corrupt the service for existing users.

We obtain optimal loose coupling when a service is a perfect black box, in the sense that any user cannot see inside it; and, when once inside the service, you cannot see out; so that the service has no idea of who is using it and what other services are around. Making sure that services really are black boxes really does reduce coupling.

Each SOA service should have a quality of service (QoS) associated with it. Typical QoS elements are security requirements, such as authentication and authorization, reliable messaging, and policies regarding who can invoke services. QoS statements may include SLAs, but in principle they can include more than mere service level agreements. I will give an example of this.

Let us suppose that our quotation service needs to vary the price of widgets dynamically as the open market price of copper fluctuates. This price must be obtained from an on-line information provider such as Reuters or Honest John's Prices Inc. as shown in Figure 1-8. Honest John provides a low cost option but, let's face it, probably isn't anywhere near a reliable as a reputable or well-established firm such as Reuters or

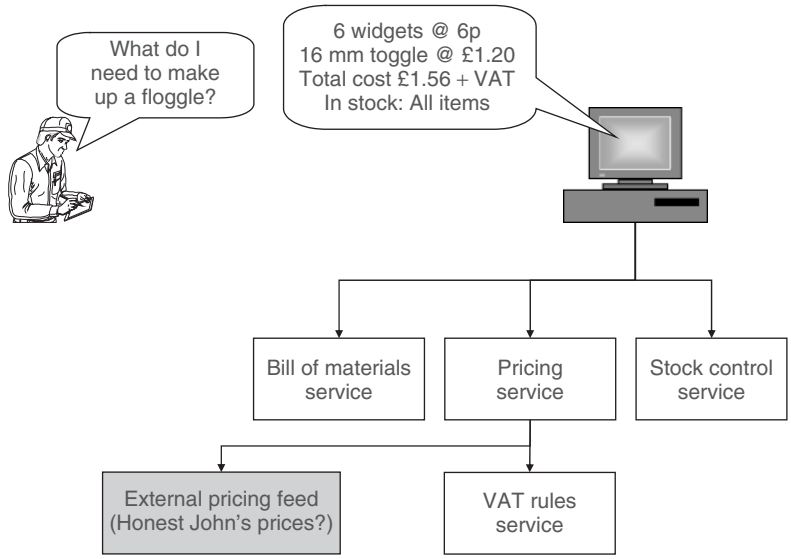
What do I need to make up a floggle?
6 widgets @ 6p
16 mm toggle @ £1.20
Total cost £1.56 + VAT
In stock: All items
Bill of materials service
Pricing service
Stock control service
External pricing feed (Honest John's prices?)
VAT rules service

Figure 1-8 External services must be trustworthy.

Thompson (Datastream). We must not know or care about how such a service is implemented but we might well want to specify a level of 'trust' or some surrogate for that quality.

There are other, more mundane, meanings for QoS. The term is most often used to refer to qualities like security or the reliability of message delivery. These are important, of course, but a good service needs to be trusted in every sense of the word.

Application developers or system integrators can build applications by composing one or more services without knowing the services' underlying implementations. For example, a service can be implemented either in .NET or J2EE, and the application consuming the service can be on a different platform or language.

SOA, if done properly, should lead to interfaces that are about the business and not about supporting the user interface; there is end-to-end involvement with both customers and suppliers. Beware especially of just wrapping existing interfaces; this is usually far too low level. Make sure, when modelling, that you understand the business, not just the computer systems. Provide services for *people* to do tasks that deliver them value, whether these people are employees, customers, suppliers, regulators or any other kind of stakeholder. Furthermore, they should be able to understand the system in their own terms rather having to learn the software developers' argot.

SOA should provide services that help people carry out tasks that deliver them value; systems for the user not for the IT department; systems for

employees, customers, suppliers, partners and so on. Most importantly, SOA must provide systems for the 'real' user.

1.3.1 The Real User

Traditional, detailed use cases define the interactions among the actors, the direct users who adopt rôles in relation to a system. In several works, Graham (1994, 1996, 1998) reinterpreted use cases to mean goal-oriented conversations or business processes, but the overwhelming number of UML practitioners followed the advice of the creator of the use case (Jacobson, 1992) in restricting use cases to those that involved a human actor or other device interacting with a system at its boundary. But the real user of the system may be far outside the system boundary. The best that could be done in this tradition was to show so-called 'business use cases' at the boundary of a bigger system that incorporated the direct users as components, leading to the kind of Russian doll topology illustrated in Figure 1-9. Is it really credible that all businesses structure themselves in this neat, nested fashioned. I don't think so.

Also, common practice was to create very detailed and over-documented use cases. At the start of the development, if the level of use case captured is too detailed then the analysts are effectively designing the user interface – this is an activity that we can often leave until later. Capturing such detailed use cases leads to vast amounts of documentation with little understanding of the business and its processes. This is exacerbated by anally-retentive project management that insists on a ream of filled-in use cases templates copied from the nearest volume of 'guruware'. As we shall see later, much of the information on the typical template (Cockburn, 2000;

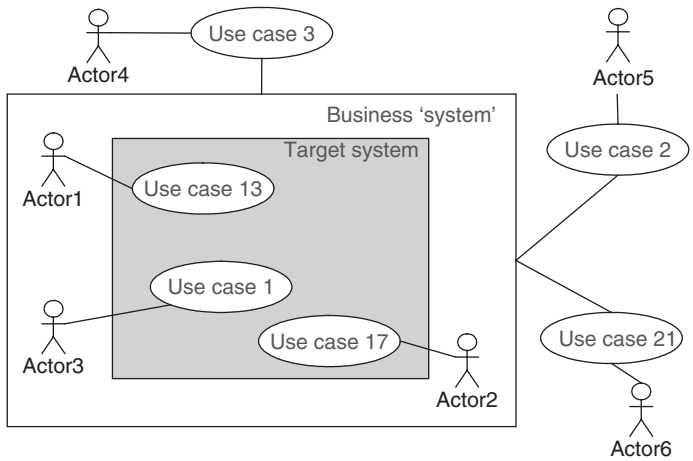

Figure 1-9 Nested use case models.

Fowler, 2003; Jacobson, 1992) is quite redundant from a business analysis viewpoint. There is, however, a need to document the description of the business at a level of detail that provides an understanding of the business. As we will learn, it is 100 % sufficient to do this by writing only the pre- and post-conditions of the use case and noting any possible exceptions, fatal and non-fatal. We return to this in Chapter 5.

In our approach, and in a nutshell, the idea is to start by documenting 'big' use cases: interactions among the real users of the system. These big use cases are business processes. At a later stage in the development cycle these big use cases will have detail added to give traditional use cases which can be used for the development of the HCI as part of the design phase.

The real user will need to be supported with services; the system user is supported by a user interface. Low level stepwise use cases are about defining the system user interfaces. It should be noted that sometimes the real user of the system is a direct user; but it is still better to find his/her goals with respect to the business rather than with respect to the system.

For example, in a lending library the real users of the system are the library members as well as the librarians (who are direct users of the library administration system). A member wants a reservation to be fulfilled when it is their turn in the queue of library members waiting for a particular book. When they return a book at the end of a loan, they want the transaction recorded so they are no longer responsible for it. Our library members are not that interested in the actual loan being recorded – though the library is, of course, very interested in that particular transaction. Thus, the purpose of the system is to make certain that the business of the library is run properly; that the business processes of the library work correctly.

For another example, consider the refuelling of planes at an airport. The technician carrying out the refuelling will be given instructions on how much fuel should be loaded onto the plane. He might contact a clerk to get the latest instructions that may vary the fuel load.

In this example, the business goal is to load the right amount of fuel onto the plane to enable it to get to its destination safely. (Typically it should arrive at its destination with enough fuel to supply a safety margin – at the other extreme, if everything that can go wrong does, the plane should still arrive safely on the ground in one piece with an empty tank!)

The main message here is that you need to get outside of the system boundary to identify services. The writing of use cases at the system boundary tells you the services the interface designer needs, to support the clerk sitting at the computer, not the real user. By concentrating on the user interface rather than what is happening in the real world, we are focusing on the solution rather than on the problem. We need to move our perception out a level (and up a level of abstraction).

There is a change of emphasis. To find business services, we need to understand the needs (goals) of the real user and supply services to satisfy those needs. We could allow the real user, the technician, to access the system directly with little or no change to the service.

A more familiar example of the real user and the actual user being different is a call centre; the real user is the customer on the end of the phone, not the call centre clerk. The software, however, is usually designed for the clerk; this may explain why call centres are not always very helpful! Note that, when we come to services we will differentiate between business services and utility services; utility services are about satisfying the goals of the developer.

Once services have been identified, the users of a service can be changed and their responsibilities moved around, sometimes radically. Returning to the example of refuelling an aircraft, this might be identified as an outsourcible service and moved out to an external service provider. The business service of refuelling an aircraft uses many other services; to calculate the fuel load of the aircraft, we will need to interrogate:

- the flight schedule, to get details of the destination, route, passenger load, and cargo load;
- the runway allocation services of the departure and arrival airports, to find out the length of the runways;
- get weather information for the complete route, especially at the departure airport.

To order the refuelling we will need:

- the arrival time at the airport;
- the gate number;
- the fuel load.

To schedule the refuelling we will need to tell the fuel company:

- the gate number;
- the refueling time;
- the amount of fuel.

If all of this information is available from various services then we can change the way we do business, to become more efficient. It is much easier for the refuelling service to schedule its fuel bowsers if it can find out when customer aeroplanes actually arrive; the fuel load can be calculated in real time and, for example, cargo can be swapped more profitably for passengers during plane reloading, and the fuel load adjusted accordingly.

Any business process can be outsourced if we view it in the right way, though we may not wish to do this. With an SOA based system, there can be end-to-end involvement with employees, customers, suppliers, etc.

For example, in the air-transport industry:

- we can use other companies' services to order limousines to move passengers from home to the airport and back again (similarly, at their destination);
- we can schedule a courier company to pick-up and deliver the luggage – using our airline if possible;
- we can use the airport system to track passengers and their luggage (RFID tags are being used for this currently);
- the fuel loading can be done just before take-off (the fuel company sorts out the fuel load based on current conditions) and we could add passengers to the load if there is room, removing less profitable non-urgent cargo;
- the aircraft maintenance company could call planes in for routine service, and provide early warning of emergency service requests;
- we could book hotels at the destination and order rental cars, theatre-tickets, tours, and provide even more services to enhance 'the passenger experience' – and make money!

The idea is to cut out the middle men and thus improve our business efficiency and/or flexibility.

A typical approach to services (in all meanings of the concept) is for the client to work out what they want and then make a request to the supplier. In our airline example, he/she might say, 'load my plane with 3 tonnes of fuel; it will be at gate 19 at 15:00'. This means that the supplier has just enough information to satisfy the request, but has little or no access to additional information that could make his/her service more efficient.

If the service supplier has access to more information, they might be able to carry out the task more efficiently. In the airline example, the fuel supplier has full access to the airline's and other services and could probably make better decisions on how to supply their service. For example, they could supply less fuel than is ordered because it has more up-to-date information about the actual fuel requirements, or it could schedule its fuel trucks more efficiently.

It can be seen that there are savings to be made, and these are often best shared between the client and the service supplier.

1.4 Business Drivers for SOA

I have already mentioned the high IT project failure rate as a compelling reason for doing *something* differently but there are other key drivers.

The main driver, in my opinion, is the need for greater business agility: the need to innovate new business processes; the need to conform to rapid changes in regulation; the need to exploit new economies of scale and reuse as opportunities arise. Exploiting and integrating the legacy in a flexible way (EAI) can clearly contribute to business agility.

In addition to this there is a need for greater business focus, a focus on the real customers for competitive edge and, finally, the need for greater consistency arising from sharing common services.

The move to B2B over XML from traditional EDI (Electronic Data Interchange) has also been a driver in some sectors but it is now less urgent since low entry-cost EDI products became available.

Some drivers cross the business–technology categories. Rapidly evolving or poorly understood requirements has been a problem for many years, as has the phenomenon of requirements creep. We have already considered the need for a common language. Involving the business in service definition and delivery. The key thing is to make sure that it isn't the lunatics that are running the asylum. High IT system maintenance costs tie up budgets that could be better used for supporting new business initiates; so that, if SOA can reduce these costs through reuse or – more likely – greater flexibility, this is a strong business argument for SOA. Finally, rapidly evolving technology and platforms present as much of a problem to the business as they do to the IT function.

1.5 Technology Drivers

One of the historical drivers that led up to the widespread interest in SOA was the need to get disparate legacy and newer systems to interoperate or be accessed through a single login. This is the so-called Enterprise Application Integration (EAI). Typically, a large organization would, over the years, have accumulated a goulash of departmental systems and systems transferred through mergers and acquisitions. To link these together various point-to-point connexions would have been established, usually based on the interchange of comma-delimited files over batch tape or FTP links. The resulting spaghetti connexions are suggested by Figure 1-10.

Various technologies have been advanced to address this problem including object request brokers (ORBs) based on the CORBA standard (Common Object Request Broker Architecture) and message-oriented middleware (MOM). The main difference between the two is that ORBs deal with messages synchronously and MOM does so asynchronously. In plainer terms, in an ORB environment, an application sends a message to another application and waits for a reply. If the systems crash, everything will have to start all over again. With MOM, such messages are placed on a queue

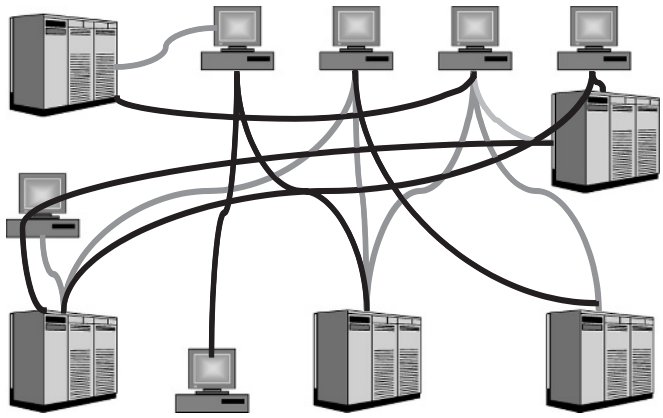

Figure 1-10 Enterprise spaghetti.

for delivery and, if necessary, the sending application can carry on with its business. Usually, eventual delivery of the message is guaranteed by the middleware. Some modern MOM products also include ORB capabilities.

Typical mainstream MOM products include IBM's MQSeries, BEA's Tuxedo, TIBCO, SonicMQ and Microsoft MQ.

Enterprise Service Buses (ESBs) build on MOM to provide a flexible, scalable, standards-based integration technology for building a loosely coupled, highly-distributed SOA. ESBs contain facilities for reliable messaging (*á la* MOM), web services, data and message transformation, content-based 'straight through' routing. Along with web services, ESBs are proving to be the major technical enablers for actual SOA projects.

The architectural simplification provided – at least potentially – by an ESB is illustrated in Figure 1-11. All we need to do is create interfaces or

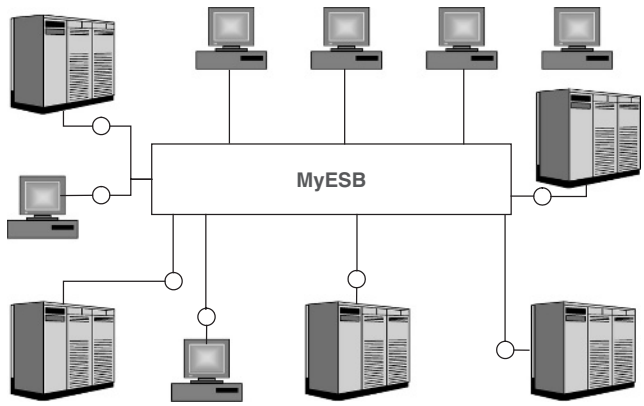

Figure 1-11 The Enterprise Service Bus.

APIs to the systems and the bus takes care of message formats and routing. Well it can if you know what you are doing!

Typical mainstream ESB vendors include BEA, Cape Clear, IBM, SeeBeyond (Sun) and Sonic (Progress).

ESBs are designed to replace conventional 'hub and spoke' EAI brokers and to compliment or replace application servers, such as Websphere and WebLogic.

Another technology that is very pertinent to SOA is that of business rules management systems (BRMSs). This technology allows systems to store business rules separately from raw computer code and in a form that can be maintained directly by business experts with little or no IT skills. Decision processes based on rules can be presented as services within an SOA. I shall have more to say about this in later chapters, when we look into business process modelling, because there is a strong connexion between the two.

Here typical mainstream products include JRules, Blaze Advisor, HaleyRules, PegaRules and Versata.

As I have repeatedly emphasized, the key technology to master when embarking on an SOA project is modelling. Modelling is core to defining good, sharable, reusable, composable, autonomous services. This book will explore how to model requirements, system specifications and business processes. Its companion volume explores modelling within the context of project management – specifically agile software development processes and business rules management.

The final core enabling technology for SOA that I want to mention is standards and, in particular, web services standards.

Web services offer a practical, standardized way of interoperating amongst applications, regardless of the platforms on which they are running. Concrete agents communicate by passing messages that conform to defined standard protocols. The environment is open, so that agents can leave or join without disrupting the whole. Web services standards are myriad but the most important ones have been XML, WSDL, UDDI and SOAP. WS-* refers to a whole family of standards for things such as security, reliable messaging, and so on. The point is that these standards have meant that users and vendors can co-operate to exchange service definitions and realizations and safely pass messages around a corporate or even external network.

TCP/IP is the basic message transport mechanism. XML is a mark-up language that can have tags to define, identify and manipulate structures representing things like customers, products and standard business transactions. WSDL (Web Services Description Language) is machine and programming language independent. SOAP (originally standing for Simple Object Access Protocol) is a communication protocol for exchanging information between applications. It can run over HTTP, JMS or other proprietary middleware. SOAP builds on XML and uses WSDL.

UDDI (Universal Description, Discovery and Integration) is a 'worldwide' business service registry. It too builds on WSDL.

This book is not about the details of these enabling standards but we will look at them in the context of SOA projects in Chapter 8. There are also important emerging standards for business process modelling that we will consider in detail later, primarily BPEL, BPMN and UML. In the context of business rules (mentioned above) the important standards are OWL, SVBR and (again) UML.

Lastly, there is a multitude of extant and emerging vertical industry standards based on XML. These mostly define the semantics of messages within the business contexts that they address. Typical examples are the domain models for the general insurance business (ACORD) and another for Oil and Gas, which was, I think, one of the earliest established. Some business rules management system vendors offer business rules 'knowledge packs' based on these standards.

Web services have self-describing interfaces in platform-independent XML documents. Web Services Description Language (WSDL) is the standard used to describe the services, at least partially (no semantics).

Web services communicate with messages defined in a language (possibly XML). Communication among consumers and providers or services can happen in heterogeneous environments, with little or no knowledge about the provider. Messages between services can be viewed as key business documents processed in an enterprise.

Web services are maintained in a registry that acts as a directory listing. Applications can look up services in the registry and invoke them. Universal Description, Definition, and Integration (UDDI) is the standard used for service registry.

But quality? Trust? Supplier reliability? Unfortunately, UDDI is not currently expressive enough to encase these factors.

1.6 Benefits, Pitfalls and Prospects

The potential benefits of adopting service oriented architecture may be summarized as follows.

- Faster development.
- Faster maintenance.
- More reusable business logic.
- Greater consistency across the enterprise.
- Better alignment and understanding between business and IT.

We shall deal first with the business benefits arising from SOA, then the technical ones.

SOA offers the possibility of being able to plug in new services without disrupting existing operations (business agility). This is especially true if one is using an enterprise service bus. Modularity is based on business concepts rather than technical models, so that business goals should be better supported. Services can be shared across organizational units, companies and maybe even geographical regions, leading to both greater consistency and, potentially, more reuse. Services can be delivered on the web. Finally, there is the opportunity to share your savings with your partners (and vice versa, of course).

The technical benefits include a better separation of concerns, an improvement in the way software is built, leading to lower maintenance costs (as we shall see in the next chapter), greater productivity, greater modularity and more code reuse. Building or assembling services based on small reusable components both supports and depends on the adoption of a culture of good design. Finally, building shared understanding and a common language should lead to less conflict and better business alignment.

Economic benefits include achieving a better understanding of the business. Often, changing business rules or processes (the order in which you do things) can simplify things and thus save costs or add value. SOA supports making such changes without massive delays.

You can add value to your own business by doing things better or by using other companies' services. Also you might add value to other people's business by offering them access to your services. You might also contract out development with better control using SOA.

SOA, as I argue in Chapter 2, ought to decrease the costs associated with both development and maintenance and all within a faster development cycle, as I argue in a companion volume (Graham, 2009).

1.6.1 Pitfalls

However, there are some pitfalls to watch out for too. New challenges for those adopting SOA include security and versioning. Quality may be more of a problem than with earlier systems, though making QoS more explicit could be seen as a benefit. Testing services may need new approaches and new tools are emerging, e.g. Parasoft, SOAPSonar, etc. With services, errors are visible globally – and customers may desert because of even one bad service experience. And, of course, there is a possibility of unforeseen legal problems and the ever-present governance issues such as technical politics and reuse management.

None of these problems is insuperable. The biggest dangers are those I have already warned of:

- Modelling systems and not the business.
- Ignoring the real users.

- Abstractions at too low a level.
- Legacy wrapping as opposing to business service design.

In addition to these there is a danger of vendor (or product line) lock-in preventing unforeseen variation. This is especially true if you are adopting an enterprise service bus (ESB). If you know what such products are capable of and use them well this shouldn't be a problem. But if you go for the quick fix implementation using an ESB, there is every danger that you will end up with hidden spaghetti (in the ESB).

To illustrate this, consider Figure 1-12 and compare it with Figures 1-10 and 1-11.

To avoid this kind of disaster we need considerable skill in modelling, we need to pay attention to architectural layering and to simplifying and rationalizing our technology base technology. Then, maybe, one needs to use a top flight ESB product and understand the full range of its capabilities. Use high-level self-describing messages; embed data format translations; use rule-based routing.

1.6.2 Post-SOA Benefits

Once we have created and bedded down an SOA, what benefits should we be able to see?

We might see that services have enabled us to change who does what or even to have reengineered the business so that we have:

- stopped doing old unprofitable things;
- installed more efficient business processes;
- obtained more business process reuse;

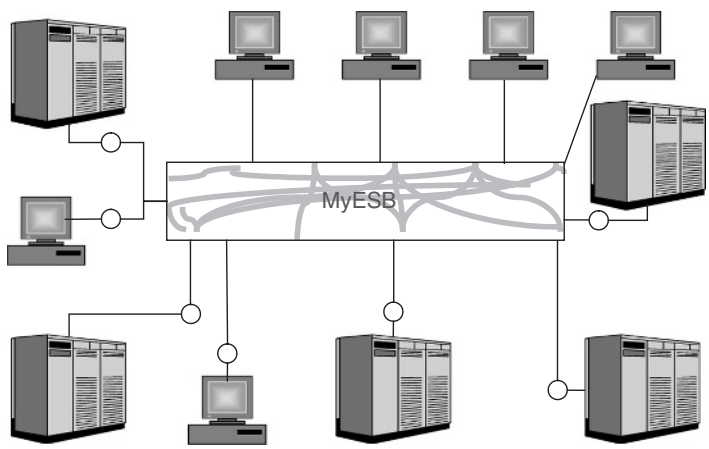

Figure 1-12 Hiding the spaghetti in the ESB.

- obtained the concomitant cost reductions;
- focused more on the business, as opposed to sexy software.

If SOA was done right, it should now be possible to keep on innovating in these dimensions as our business and markets evolve. Business process improvement is now normal practice.

SOA should have helped with granularity by giving better modularity of business processes and the services associated with them.

Building new systems on top of services – system assembly from components – is a routine part of everyday life, rather than a cause for sighs and headshakes in the 'application backlog' department.

It is now possible to share services with customers and suppliers. You are almost certainly contracting out some services now. Maybe you are contracting in some services too; doing new things by using other peoples' services. It may even be possible to use published services from other businesses, sharing pre-competitive expertise with the market, though, it has to be said that most current SOA projects are 'behind the firewall'. We will discuss this further in Chapter 3 when we consider UDDI.

1.7 Migration Strategies

As with any technology transformation programme, one of the problems that is likely to divert or derail the process is change management. Hearts and minds must be won from the IT teams, through the business units and right on to the senior business management.

Looking at change programmes that have succeeded, it seems that the first thing to secure is the understanding and strong support of the most senior operational management. Securing management 'buy in' is not only difficult but crucial. You do not start this by asking them if they 'want an SOA'. The benefits must be discussed entirely in terms of ROI, margins, shareholder wealth, markets and business processes.

Start with manageable but visible projects; no one will notice if you fail with a no-so-visible project – but no one will notice if you succeed either. The early projects should be able to demonstrate a tangible business benefit. Some 'quick wins' will help cement business support and assist the process is change management process greatly.

As I argued at the outset of this chapter, rebuilding trust between IT and the business is a key challenge and may not be easy if the atmosphere has been sullied by past failures. Therefore, be agile: don't resist changes to requirements mid project; learn how to descope rather than run late. Fix the business objectives by all means, but don't fix the specification. Test as you go to avoid unpleasant surprises during user acceptance testing.

Education and training will be important. Teaching the IT guys how to talk plain English is a good start!

On the technical side of change management, define the long-term architecture and define the (local) domain model. Make sure you are *all* clear about the software development method you are following. Publish the models, method and architecture. The companion volume on agile processes provides explicit techniques for these issues.

Finally, manage expectations. Do not let people assume that the existence of a working prototype means that all technical and business issues have been resolved. Warn your sponsors that success with service oriented architecture is a long-term investment yielding long-term benefits – though, of course, one may hope for short-term wins too. An SOA is not just for Christmas!

In my opinion, the critical success factors for SOA are these.

- Senior management support.
- Users who are prepared to get involved.
- A watertight business case for any pilots that are contemplated.
- Good modelling skills on the IT side. These should include UML, business process modelling and domain and service modelling skills.
- You have thought (i.e. done the modelling) before deciding to buy infrastructure products such as an ESB.
- Your team has good (non-IT) communication skills.
- A reuse culture is at least conceivable in the IT department.

  . . .and, right down at the bottom of the priority list

- The requisite technical and product skills are in place or obtainable.

If any of these are not present then question whether this is the right time (or the right company) to start an SOA project in.

1.8 Summary

Too many software projects fail. Software engineering practice has not delivered on its promises and needs to change. The chief problem is how best to align IT practice with business need.

Requirements engineering is mainly a modelling problem as is specification. But they are *not* the same problem. The requirements engineer must look beyond the system boundary and understand the business and the real users of services. Discovering business rules is an important part of requirements engineering.

Service oriented architecture is an architectural concept in software design that emphasizes the use of combined loosely coupled services

to support business requirements directly. In SOA, resources are made available to service consumers in the network as independent artifacts that are accessed in a standardized way. This adherence to standardization is definitional. SOA is about raising the level of abstraction, so that requirements and business processes can be discussed in a language understood by business as well as IT people. The main idea behind SOA is the desire to build applications that support a business process by combining a number of smaller business services into a complete business process.

Though built on similar principles, SOA is not the same as eb services. SOA is independent of any specific technologies.

A service is a unit of work carried out by a service provider to achieve some desired result for the service consumer. A service is a loosely coupled black box that can be found and (re)used as part of a current or new software application. The more loosely coupleable it is, the more (re)useable it is. Such black boxes can be assembled in different ways to do business in different ways. The IT function can then adapt to changing requirements more easily. It should also supply a service to the business; the operations that make up the service are about supporting the business, not the user interface.

Although a service is just an interface, it will thus be implemented by software components that implement a reusable business function, such managing customers or managing customer accounts. Services are defined by their interfaces, which describe what the services can do. How the service works is hidden inside the components that provide the service.

Services must be autonomous, abstract, composable, discoverable, reusable black boxes defined by contracts.

A service can receive a message, process it, return a reply and then forget all about it. Looking inside the service, however, we might see three types of communication ports or 'endpoints'.

- Ports for messages entering the service; the **entry** ports.
- A departure port for sending the reply message; the **exit** port.
- A **rejected message** port for invalid messages.

Each SOA service should have a quality of service (QoS) associated with it. Typical QoS elements are security requirements, reliable messaging and policies regarding who can invoke services. QoS statements may include SLAs, but in principle they can include more than mere service level agreements.

The main business drivers for SOA are as follows:

- The need for greater business agility: the need to innovate new business processes.
- The need to conform to rapid changes in regulation.

- The need to exploit new economies of scale and reuse as opportunities arise. Exploiting and integrating the legacy in a flexible way through EAI. This can clearly contribute to business agility.
- Rapidly evolving or poorly understood requirements.

The chief technology driver has been getting disparate legacy and newer systems to interoperate or be accessed through a single login: Enterprise Application Integration.

Enterprise Service Buses build on MOM to provide a flexible, scalable, standards-based integration technology for building a loosely coupled, highly distributed SOA. ESBs contain facilities for reliable messaging, web services, data and message transformation, content-based 'straight through' routing. Along with web services, ESBs are proving to be the major technical enablers for actual SOA projects.

The key technology to master when embarking on an SOA project is modelling. Modelling is core to defining good, sharable, reusable, composable, autonomous services. Standards and, in particular, web services standards, provide core enabling technology for SOA.

The potential benefits of adopting service oriented architecture may be summarized as follows:

- Faster development.
- Faster maintenance.
- More reusable business logic.
- Greater consistency across the enterprise.
- Better separation of concerns.
- Better alignment and understanding between business and IT.

There are economic benefits too. These include achieving a better understanding of the business; if everyone knows the rules of the business there are less likely to be conflicts. Often, changing business rules or processes (the order in which you do things) can simplify things and thus save costs or add value. SOA supports making such changes without massive delays.

The potential pitfalls include the following:

- Modelling systems and not the business.
- Ignoring the real users.
- Abstractions at too low a level.
- Legacy wrapping as opposing to business service design.
- The danger of vendor lock-in preventing unforeseen variation.

Education and training will be important in migrating to SOA. Teaching the IT guys how to talk plain English is a good start!

The critical success factors for SOA are these.

- Senior management support.
- Users who are prepared to get involved.
- A watertight business case for any pilots that are contemplated.
- Good modelling skills on the IT side. These should include UML, business process modelling and domain and service modelling skills.
- You have thought (i.e. done the modelling) before deciding to buy infrastructure products such as an ESB.
- Your team has good (non-IT) communication skills.
- A reuse culture is at least conceivable in the IT department.
- The requisite technical and product skills are in place or obtainable.

Businesses strive for shorter time to market and lower development and maintenance costs. Using business process modelling and possibly BRMSs together with better requirements engineering and business modelling within the context of SOA should decrease development costs and dramatically shorten development and maintenance cycles.

1.9 Bibliographical Notes

Erl (2005) is a very good introduction to the technology and broad concepts of SOA but does not emphasize the importance of modelling or provide guidance thereto.

Chappell (2004) is an informed discussion of ESBs by an innovator in that area but, unfortunately, its undoubted technical value is marred by very poor use of the American language, making it rather difficult to read. Chapell builds on the notation developed by Hohpe and Woolf (2004), who describe 65 patterns for developing enterprise integration projects using asynchronous messaging along with lots of useful code examples. Many of the ideas in this book are of considerable significance to those implementing SOA.

Graham (2006) provides a detailed discussion of and tutorial on business rules management systems and references to the literature on that subject. Another forthcoming companion volume to this book is Graham (2009).